CHARUSAT
CHAROTAR UNIVERSITY OF SCIENCE AND TECHNOLOGY

cspit
Charusat Shri C. J. Patel Institute of Technology

CE144
OBJECT ORIENTED PROGRAMMING WITH C++

UNIT-6
Constructors and Destructor

N. A. Shaikh
nishatshaikh.it@charusat.ac.in

Topics to be covered

- **Introduction to Constructors**
- **Parameterized Constructors**
- **Multiple Constructors in class**
- **Constructors with default argument**
- **Dynamic initialization of Constructors & objects**
- **Copy Constructor**
- **Dynamic Constructor**
- **Destructors**

Introduction

- One of the aims of C++ is to create user-defined data types such as class, that behave very similar to the **built-in types**.
- This means that we should be able to **initialize a class type variable (object) when it is declared**, much the same way as initialization of an ordinary variable.
- Similarly, when a variable of built-in type goes out of scope, the **compiler automatically destroys the variable.**
- But it has **not happened with the objects** we have so far studied.

Introduction

- It is therefore clear that some more features of classes need to be explored that would enable us to **initialize the objects when they are created and destroy them when their presence is no longer necessary.**
- C++ provides a special member function called the **constructor** which enables an object to initialize itself when it is created. This is known as **automatic initialization of objects.**
- It also provides another member function called the **destructor** that destroys the objects when they are no longer required.

Constructors

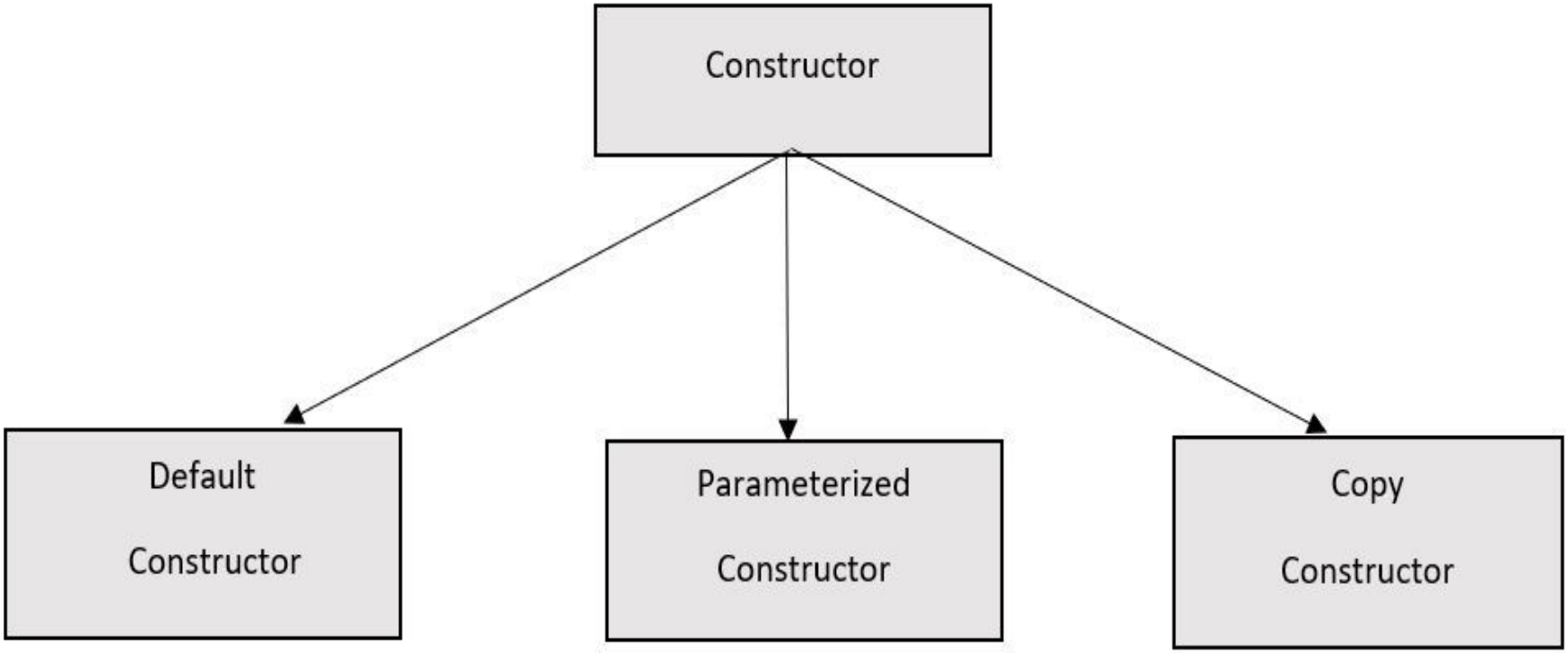
- A **constructor** is a special member function of a class which initializes objects of a class.
- It is special because its **name is the same as the class name.**
- Constructor is **automatically called/invoked** when object(instance of class) is created.
- It is called constructor because it **constructs the values of data members of the class.**
- It has **no return type**, so can't use return keyword.
- It must be an **instance member function**, that is, it can never be static.

Characteristics of Constructor

- They should be declared in the **public** section.
- They do **not have return type**, not even void therefore, they return nothing.
- They **can not be inherited** though derived class can call the base class constructor.
- Like other C++ functions, they can have **default arguments**.
- They **can not be virtual**.
- We **can not refer to their address**.
- An object with a constructor (or destructor) can not be used as a member of a **union**.
- They make 'implicit calls' to the operators **new** and **delete** when memory allocation is required.

NOTE: When a constructor is declared for a class initialization of the class, objects becomes mandatory.

Types of Constructor



Default Constructor

- **Default constructor** is the constructor which doesn't take any argument. It has no parameters

Note:

- Even if we do not define any constructor explicitly, the **compiler will automatically provide a default constructor** (expects no parameters and has an empty body) implicitly

Default Constructor

```
//class with a constructor
class integer
{
     int m,n;
public:
     integer(void);          //constructor declared
     ..............
     ..............
};

integer:: integer(void)   //constructor defined
{
     m=0;
     n=0;
}
```

```
integer ob;
```

- Above statement not only creates the object but also **initializes its data members m and n to zero.**
- There is no need to write any statement to invoke the constructor as we do with the normal member function.

Default Constructor

```cpp
#include <iostream>
using namespace std;

class construct
{
    int a, b;
public:
    // Default Constructor
     construct()
    {
        cout<<"Default constructor called\n";
        a = 0;
        b = 0;
    }

    void display()
    {
        cout << "a=" << a << endl << "b=" << b <<endl;
    }
};

int main()
{
    // Default constructor called automatically
    // when the object is created
    construct ob1;
    ob1.display();

    construct ob2;
    ob2.display();

    return 0;
}
```

```
Default constructor called
a=0
b=0
Default constructor called
a=0
b=0
```

Parameterized Constructor

- **Parameterized constructor** is the constructor which can take argument(s).
- It is used to initialize the various data elements of different objects with different values when they are created.
- It is used to **overload constructors**.

```
class integer
{
     int m,n;
public:
     //parameterized constructor
     integer(int x,int y);
     ......................
     ......................
};

integer::integer(int x,int y)
{
     m=x;
     n=y;
}
```

```
class integer
{
     int m,n;
public:
     //Inline constructor
     integer(int x,int y)
     {
          m=x;
          n=y;
     }
};
```

Parameterized Constructor

- When an object is declared in a parameterized constructor, the initial values have to be passed as arguments to the constructor function.

This can be done in two ways:

1. **By calling the constructor explicitly**

```
integer ob=integer(0,100);  //explicit call
```

2. **By calling the constructor implicitly**

```
integer ob(0,100);  //implicit call
```

Parameterized Constructor

```
#include<iostream>
using namespace std;

class integer
{
    int m,n;
public:
    integer(int x,int y)
    {
        cout<<"\nTwo parameterized constructor";
        m=x;
        n=y;
    }

    void display(void)
    {
        cout<< "m=" <<m <<endl <<"n=" <<n <<endl;
    }
};
```

```
int main()
{
    //constructor called implicitly
    integer ob1(0,100);
    cout<<"\nObject1"<<endl;
    ob1.display();

    //constructor called explicitly
    integer ob2=integer(25,75);
    cout<<"\nObject2"<<endl;
    ob2.display();

    return 0;
}
```

```
Two parameterized constructor
Object1
m=0
n=100

Two parameterized constructor
Object2
m=25
n=75
```

Parameterized Constructor

```
#include<iostream>
using namespace std;

class integer
{
    int m,n;
public:
    integer(int x)
    {
        cout<<"\nSingle parameterized constructor";
        m=n=x;
    }

    void display(void)
    {
        cout<< "m=" <<m <<endl <<"n=" <<n <<endl;
    }
};
```

```
int main()
{
    //constructor called implicitly
    integer ob1(100);
    cout<<"\nObject1"<<endl;
    ob1.display();

    //constructor called explicitly
    integer ob2=integer(75);
    cout<<"\nObject2"<<endl;
    ob2.display();

    //nameless temporary object
    integer(800);

    return 0;
}
```

```
Single parameterized constructor
Object1
m=100
n=100

Single parameterized constructor
Object2
m=75
n=75

Single parameterized constructor
```

Copy Constructor

- The parameters of a constructor can be of any type except that of the **class** to which it belongs.

```
class A
{
      ......
      ......
public:
      A(A);      //illegal
};
```

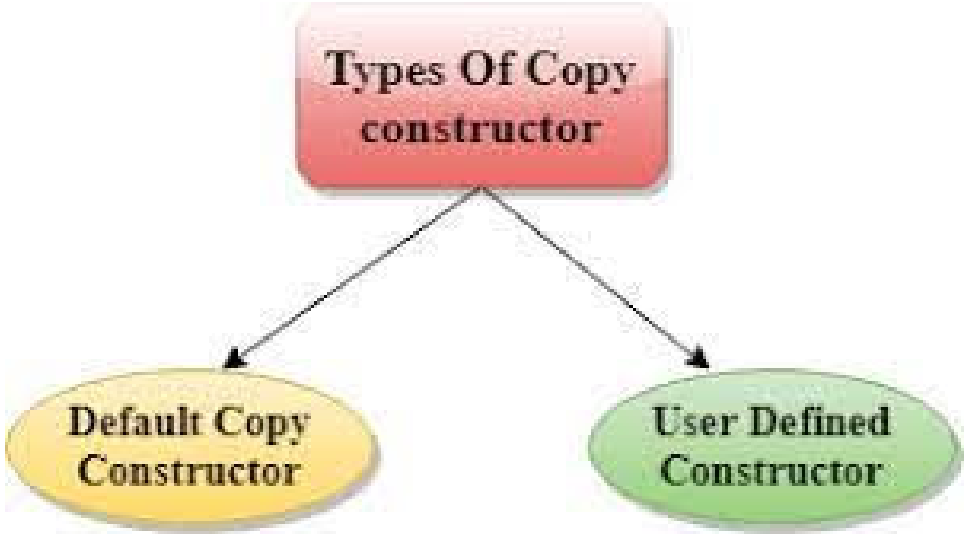
- However, a constructor can accept a **reference** to its own class as a parameter

```
class A
{
      ......
      ......
public:
      A(&A);     //Valid
};
```

- In such cases, the constructor is called the **copy constructor**.

Copy Constructor

- A **copy constructor** is a member function which initializes an object using another object of the same class.



Copy Constructor

```
#include<iostream>
using namespace std;

class code
{
    int id;
public:
    code() //default constructor
    {

    }
    code(int a) //parameterized constructor
    {
        id=a;
    }
    code(code &x) //copy constructor
    {
        id=x.id;
    }
    void display()
    {
        cout<<id;
    }
};
```

```
id of A:100
id of B:100
id of C:100
id of D:100
```

```
int main()
{
    code A(100);
    cout<<"\n id of A:";
    A.display();

    //copy constructor called
    code B(A);
    cout<<"\n id of B:";
    B.display();

    //copy constructor called again
    code C=A;
    cout<<"\n id of C:";
    C.display();

    code D;
    //Assignment operator,not copy constructor
    D=A;
    cout<<"\n id of D:";
    D.display();
}
```

Copy Constructor

NOTE:

- A **reference** variable has been used as an argument to the copy constructor.
- We can not pass the argument by value to a copy constructor.
- When no copy constructor is defined , **the compiler supplies its own copy constructor.**

Two types of copies are produced by the constructor:

1. **Shallow copy**
2. **Deep copy**

Shallow Copy

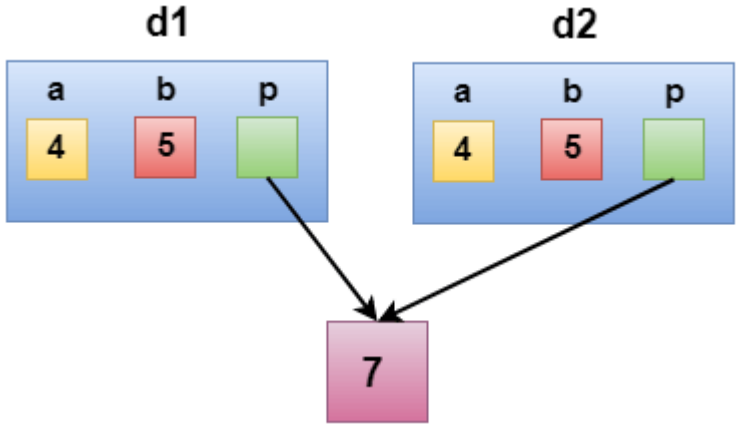
- The **default copy constructor** can only produce the **shallow copy.**

- A Shallow copy is defined as the process of creating the copy of an object by copying data of all the member variables **as it is.**

Shallow Copy

```
class Demo
{
    int a,b,*p;
public:
    Demo()
    {
        p=new int;
    }
    //Default copy constructor supplied by compiler
    /*Demo(Demo &d)
    {
        a = d.a;
        b = d.b;
        p = d.p;
    }*/
    void setdata(int x,int y,int z)
    {
        a=x;
        b=y;
        *p=z;
    }
    void showdata()
    {
        cout << "value of a is : " <<a<< endl;
        cout << "value of b is : " <<b<< endl;
        cout << "value of *p is : " <<*p<< endl;
    }
};
```

```
int main()
{
  Demo d1;
  d1.setdata(4,5,7);
  Demo d2 = d1;
  d2.showdata();
  return 0;
}
```




```
value of a is : 4
value of b is : 5
value of *p is : 7
```

Deep copy

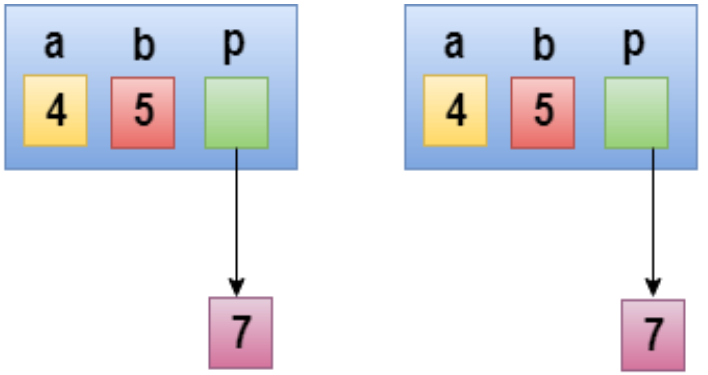
- **Deep copy** is possible only with **user defined copy constructor.**

- In user defined copy constructor, we make sure that **pointers** (or references) of copied object **point to new memory locations.**

Deep copy

```
class Demo
{
      int a,b,*p;
public:
      Demo()
      {
            p=new int;
      }
      Demo(Demo &d)
      {
            a = d.a;
            b = d.b;
            p = new int;
            *p = *(d.p);
      }
      void setdata(int x,int y,int z)
      {
            a=x;
            b=y;
            *p=z;
      }
      void showdata()
      {
            cout << "value of a is : " <<a<< endl;
            cout << "value of b is : " <<b<< endl;
            cout << "value of *p is : " <<*p<< endl;
      }
};
```

```
int main()
{
   Demo d1;
   d1.setdata(4,5,7);
   Demo d2 = d1;
   d2.showdata();
   return 0;
}
```




```
value of a is : 4
value of b is : 5
value of *p is : 7
```

Let’s Practice

1. Copy Constructor VS Assignment Operator

2. Shallow Copy VS Deep Copy

Multiple Constructor in a Class

- The process of sharing the same name by two or more functions is referred to as **function overloading**.

- Similarly, when more than one constructor function is defined in a class, we say that the **constructor is overloaded.**

Multiple Constructor in a Class

```
class Add
{
     int m, n;
public :
     Add()
     {
          m = 0;
          n = 0;
     }
     Add(int a, int b)
     {
          m = a;
          n = b;
     }
     Add(const Add & i)
     {
          m =i.m;
          n =i.n;
     }
     int display()
     {
          return (m+n);
     }
};
```

```
int main()
{
     Add A1, A2(10, 15), A3(A2);
     cout<< "\nObject A1 value : " << A1.display();
     cout<< "\nObject A2 value : " << A2.display();
     cout<< "\nObject A3 value : " << A3.display();
     return 0;
}
```

```
Object A1 value : 0
Object A2 value : 25
Object A3 value : 25
```

Constructors with Default Arguments

- It is possible to define constructors with **default arguments**

NOTE:

- It is important to distinguish between the **default constructor A::A()** and the **default argument constructor A::A(int = 0).**
- The default argument constructor can be called with either one argument or no arguments.
- When called with no arguments, it becomes a default constructor.
- When both these forms are used in a class, it causes **ambiguity** for a statement such as

A a;

- The ambiguity is whether to 'call' A::A() or A::A(int= 0).

Constructors with Default Arguments

```
class Demo
{
    int X,Y;
public:
    //Default or no argument constructor.
    Demo()
    {
        X = 0;
        Y = 0;
        cout << endl << "Default Constructor Called";
    }
    //Parameterized constructor with default argument.
    Demo(int A, int B=20)
    {
        X = A;
        Y = B;
        cout << endl << "Parameterized Constructor Called";
    }
    void putValues()
    {
        cout << endl << "Value of X : " << X;
        cout << endl << "Value of Y : " << Y << endl;
    }
};
```

```
int main()
{
    Demo d1= Demo(10);
    cout << endl <<"D1 Value Are : ";
    d1.putValues();

    Demo d2= Demo(30,40);
    cout << endl <<"D2 Value Are : ";
    d2.putValues();

    return 0;
}
```

```
Parameterized Constructor Called
D1 Value Are :
Value of X : 10
Value of Y : 20

Parameterized Constructor Called
D2 Value Are :
Value of X : 30
Value of Y : 40
```

Dynamic Initialization of Objects

- Class objects can be initialized dynamically too.
- The initial value of an object may be provided during **runtime**.

Advantage:

- We can provide various **initialization formats**, using overloaded constructors.
- Provides the **flexibility** of using different format of data at runtime depending upon the situation.

Dynamic Initialization of Objects

```
class X
{
    int n;
    float avg;
public:
    X(int p,float q)
    {
        n=p;
        avg=q;
    }
    void disp()
    {
        cout<<"\nRoll number=" <<n;
        cout<<"\nAverage="<< avg;
    }
};
```

```
int main()
{
    int a;
    float b;
    cout<<"\nEnter the Roll Number: ";
    cin>>a;
    cout<<"Enter the Average: ";
    cin>>b;

    X x(a,b );    //dynamic initialization
    x.disp();

    return 0;
}
```

```
Enter the Roll Number: 1
Enter the Average: 2.5

Roll number=1
Average=2.5
```

Dynamic constructor

- **Dynamic constructor** is used to allocate the memory to the objects at the run time.
- Memory is allocated at run time with the help of **'new'** operator.
- By using this constructor, we can dynamically initialize the objects.
- Thus, object is going to use **memory region**, which is **dynamically created by constructor.**

Dynamic constructor

```
class dynconst
{
    int a,b;
    int *p;
public:
    dynconst()
    {
        a=0;
        b=0;
        p=new int;
        *p=10;
    }
    dynconst(int x,int y,int z)
    {
        a=x;
        b=y;
        p=new int;
        *p=z;
    }
    void disp()
    {
        cout<<"\na=" <<a<<"\nb="<<b<<"\n*p="<<*p;
    }
};
```

```
int main()
{
    dynconst o1,o2(3,4,5);

    cout<<"Value of o1:";
    o1.disp();

    cout<<"\n\nValue of o2:";
    o2.disp();

    return 0;
}
```

```
Value of o1:
a=0
b=0
*p=10

Value of o2:
a=3
b=4
*p=5
```

const objects

- We may create and use **constant objects** using **const** keyword before object declaration.

```
const matrix X(m,n);   //object X is constant
```

- Any **attempt to modify the values** of m and n will generate **compile-time error**.
- Further, a constant object can call only **const member functions.**
- As we know, a const member is a function prototype or function definition where the keyword **const** appears after the function's signature.
- Whenever const objects try to invoke **non-const member functions, the compiler generates errors**.

Destructor

- A **destructor** is used to destroy the objects that have been created by a constructor
- Like constructor, the destructor is a member function with the **same name as the class name** but is preceded by a **tilde**

```
~integer()
{

}
```

- A destructor **never takes any argument** nor does it return any value
- It will be **invoked implicitly by the compiler** on exit from the program or block or function as the case may be to clean up storage that is no longer accessible

Destructor

- It is a good practice to declare destructors in a program since it **releases memory space for further use**
- Only **one destructor** is written to destroy all the constructors
- Whenever **new** is used to allocate memory in the constructor, we should use **delete** to free that memory
- The primary use of destructors is to free up the memory reserved by an object before it gets destroyed
- Destructor will be called in **reverse order** of the constructor creation

Destructor

```
#include<iostream>
using namespace std;

int count=0;

class alpha
{
public:
    alpha()
    {
       count++;
       cout<<"\nNo. of object created "<<count;
    }

    ~alpha()
    {
        cout<<"\nNo. of objects destroyed "<<count;
        count--;
    }
};
```

```
int main()
{
    cout<<"\nEnter Main";

    alpha A1,A2,A3,A4;

    {
        cout<<"\n\nEnter Block1";
        alpha A5;
    }

    {
        cout<<"\n\nEnter Block2";
        alpha A6;
    }

    cout<<"\n\nRe-Enter Main";
}
```

Destructor

```
Enter Main
No. of object created 1
No. of object created 2
No. of object created 3
No. of object created 4

Enter Block1
No. of object created 5
No. of objects destroyed 5

Enter Block2
No. of object created 5
No. of objects destroyed 5

Re-Enter Main
No. of objects destroyed 4
No. of objects destroyed 3
No. of objects destroyed 2
No. of objects destroyed 1
```

Destructor

```
class X
{
    int n;
public:
    X()
    {
        cout<<"\nDefault constructor called";
        n=0;
        cout<<"\nValue of n: " <<n;
    }
    X(int p)
    {
        cout<<"\n\nParameterized constructor called";
        n=p;
        cout<<"\nValue of n: " <<n;
    }
    X(const X &x)
    {
        cout<<"\n\nCopy constructor called";
        n =x.n;
        cout<<"\nValue of n: " <<n;
    }
    ~X()
    {
        cout<<"\nDestructor called\n";
    }
};
```

```
int main()
{
    X(100);
    X x1, x2(50), x3(x2);
    return 0;
}
```

```
Parameterized constructor called
Value of n: 100
Destructor called

Default constructor called
Value of n: 0

Parameterized constructor called
Value of n: 50

Copy constructor called
Value of n: 50
Destructor called

Destructor called

Destructor called
```

Practical 23: Constructor and Destructor

Write a C++ program having class time with data members: hr, min and sec.

Define following member functions.

1) getdata() to enter hour, minute and second values
2) putdata() to print the time in the format 11:59:59
3) default constructor
4) parameterized constructor
5) copy constructor
6) Destructor.

Use 52 as default value for sec in parameterized constructor.

Use the concepts of **default constructor, parameterized constructor, Copy constructor, constructor with default arguments and destructor**.

Practical 23: Constructor and Destructor

Practical 23: Constructor and Destructor

Practical 23: Constructor and Destructor

End of Unit-6